ACM system documentation

Romain for the electronics and DAQ group

July 18, 2024

Abstract

The Acquisition and Control Module system described here is a complete system to acquire image from skipper CCD dedicated to the DAMIC-M experiment with more than 50 CCD module but also to smaller CCD setup comprising one or several CCD module. The following document includes the required information to install, operate and understand this system.

1 System description and installation

The system is composed of a hardware part which includes two electronics boards for an individual system and one synchronisation board when several system are operated, a dedicated firmware, and a software suite on a PC.

1.1 Hardware description

Minimal system An individual 1 hardware system is composed of:

- an ACM board (Fig. 1 left) `https://edg.uchicago.edu/~bogdan/DAMIC_ACM/index.html`
- a front end board (FE board) 1 (Fig. 1 middle) `https://gev.uchicago.edu/cgi-bin/DocDB/ShowDocument?docid=1117`
- a SAMTEC cable (ERCD-030-120.0-TBR-TTL-1-C)
- an ethernet cable to connect to the PC

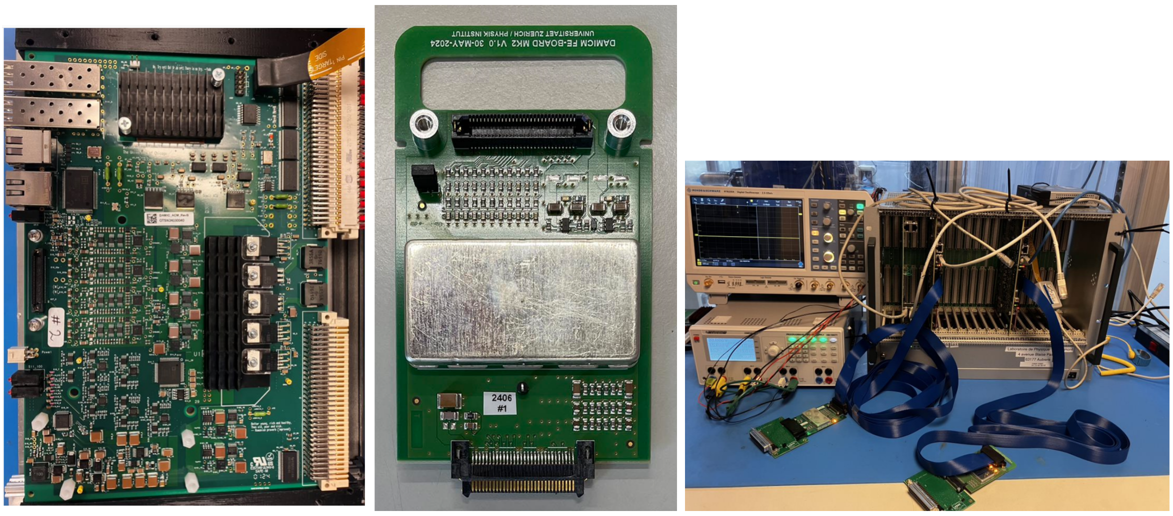


Figure 1: Left: ACM board. Middle: FE board. Right: complete system with two ACM, FE board, one synchronisation board and adapter board

This system has to be connected to a chamber with a CCD or a CCD module that can have a JFET installed or not. If one has to convert the SAMTEC connection to a subD 50, adapters board were designed for this purpose [1].

Power supplies The ACM board is powered by VME crate which provide 3.3V and 5V. The additional required voltages (+-15V and -30V) are provided by an external power supply that should be connected to the user defined voltages (+V1, -V1, +V2, -V2) of the VME standard. The connection is usually made at the back of the VME crate on a dedicated connector or pins. The expected voltages are the following [2]:

- +V1 = +15V
- -V1 = -15V
- -V2 = -30V
- +V2 = NC

Synchronisation When two or more boards are operated at the same time, one can synchronise them thanks to an additional board that should be installed in a VME for the power supply. The synchronisation signals are transmitted with ethernet cable to the top RJ45 connector of the ACM.

Installation The installation should be straighforward. The ACM board is plugged in the back plane of the VME crate. The SAMTEC cable connects the FE board to the ACM.

Tip 1: if the ACM has no extractor, you should hear and feel a click when the board is well inserted, otherwise you can have a loose connection

Tip 2: pay attention to the SAMTEC cable orientation

1.2 Firmware

The firmware should be already installed on the board and running at power up. The main blocks are:

- an ethernet block to handle the UDP communication with the PC
- a sequencer block. This part is inherited from LSST (see here [3] for a complete documentation)
- an ADC block to get the data out of the ADC chip
- a data processing block to average the ADC samples given the integration time and output the data in the chosen format described in section 2.2.1.

installation In case one needs to flash a new firmware version, you will need a USB blaster and an installation of the quartus programmer software [4]. The procedure to install the software is described in the Readme of this github link: `https://gitlab.in2p3.fr/damicm/ldaq/-/tree/ACM_UChicago_pip?ref_type=heads`.

1.3 Software / Computer

1.3.1 Computer

requirements

- PC with linux (centos, ubuntu, debian)
- ethernet network card of 1GB/s minimun
- disk space > 50 GB (depending on the data that will be acquired)
- ethernet switch for several ACM (1GB/s minimum)

settings

- IP address: set an ip address corresponding to the boards you are reading. The ACM boards are numbered from 101 to 152. If you want to operate board 106 for instance, you should add the following fixed IP address to your network connection with this command:

```
sudo ip addr add 192.168.106.1/24 dev <interface>
```

- network card and linux buffer setting.

```
ethtool -g <interface> # to check the current setting
sudo ethtool -G <interface> rx 4096 # to change it to 4096
sudo sysctl -w net.core.rmem_max=100000000
sudo sysctl -w net.core.rmem_default=100000000
```

- firewall settings

```
sudo firewall-cmd --permanent --zone=trusted --change-interface=<interface>
sudo firewall-cmd --zone=trusted --change-interface=<interface>
```

1.3.2 Software

The software is composed of two levels: on one side the central DAQ (CDAQ) which is the user interface and a server that allows to control several systems and one the other side one or several local DAQ (LDAQ) which is are client to the CDAQ server and are specific to one ACM system.

CDAQ The CDAQ can be viewed as a central broker that centralised the requests from the users, distribute them to clients in our case mostly CCD system (LDAQ) and conversely collects information from the client and serves them back to the users or other services (see the sketch on Fig. 2).The CDAQ receives commands from the user addressed to a specific client and puts them in a queue to be executed. There are two types of queue a simple one and a repeat queue. The simple queue has precedence. The CDAQ waits for the command completion before executing a new one. One can group clients into '*pool*' to address the same command to multiple clients. In practice this is how we operate several CCDs.
The installation steps and the operation are explained on the github page: `https://gitlab.com/nicolaseavalos/ccd-cdaq`

Central DAQ

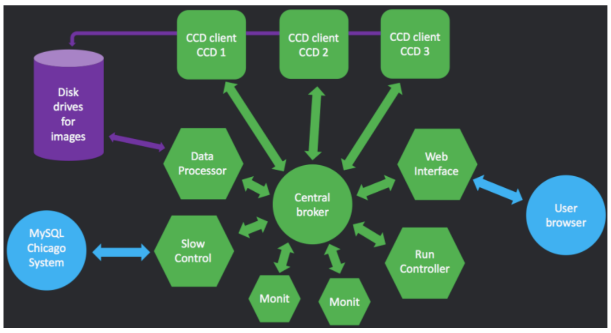


Figure 2: CDAQ sketch

LDAQ The LDAQ is the code that contains the commands to operate one ACM board (or a CCD system). It is itself composed of:

- one low level part (or backend part) written in c++ that communicates directly with the ACM board. This part makes the link between the ACM board and the frontend. It also collects the data from the computer buffer and write them down into binary files.
- a higher level part (or frontend part) written in python that contains the human readable instructions such as '*startup_ccd*' or '*take_image*' or '*erase_ccd*'. Those functions are the ones used by the CDAQ.
- a decoder that converts the binary files into FITS or csv (for raw data). Note that the decoder is NOT run automatically when acquiring an image. One needs either to run it manually or to use the CDAQ to automatically do this job.

The code is available on Gitlab: `https://gitlab.in2p3.fr/damicm/ldaq`. See the version in paragraph 1.3.2. You should download the code in the CDAQ folder.
note: you should compile the two codes in those folders: acmpy/acmdaq/ and acmpy/decoderlib/

software versions This part is likely to evolve, check the date on top of the document to see if it is not too old.

- CDAQ: v1.x
- LDAQ: tagged V2

2 Operation

At this stage, you should have a CCD and an electronics system connected together. The ACM board is connected to your computer through an RJ45 cable .

2.1 Hardware

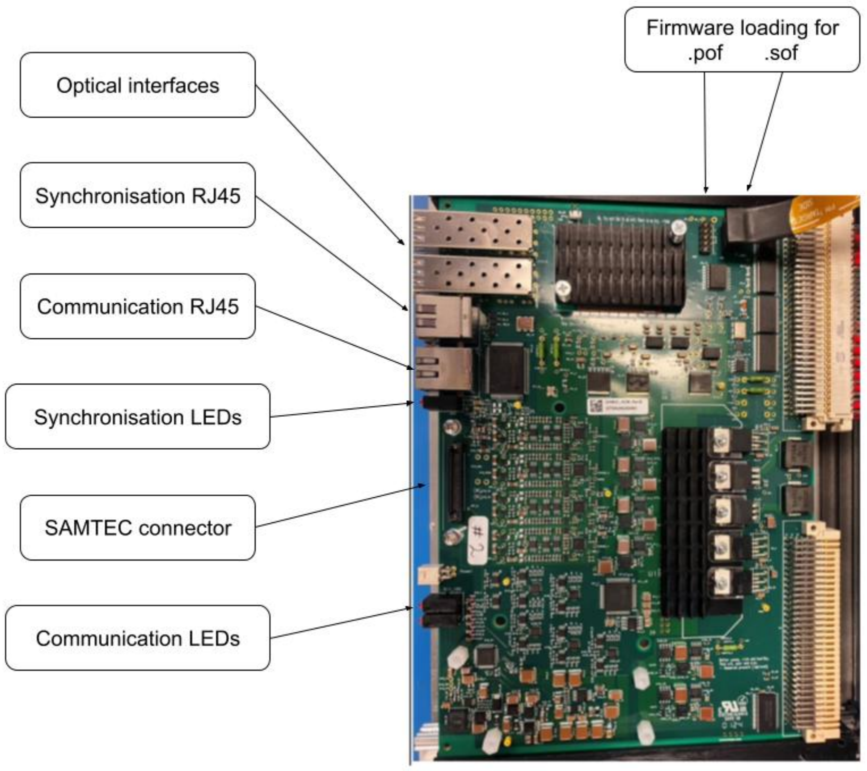


Figure 3: Connectors and LED identification for the ACM

ACM board

power supplies There is a power supply startup procedure:

1. -30V,
2. +- 15V together
3. the VME Crate (i.e. 3.3V and 5V)

The shutdown procedure is oppposite. Note: don't freak out if you don't do it right, in paris we have been using another procedure and didn't find any issue.
If you have a way to measure current, the consumption for one system should be approximately:

1. 3.3V: 3A
2. 5V : 1.5A
3. +15V : 400mA
4. -15V : 400mA
5. -30V : few mA

LEDs

- Communication LED: They blink when the firmware is correctly loaded, I don't know the exact meaning though.
- Synchronisation LEDs: ORANGE ON = external clock, RED ON = Master, GREEN: ON = Slave.

Synchronisation If you have several boards and a synchronisation board, you should connect the synchronisation board to the synchronisation RJ45 connectors of the ACM. The synchronisation provides an FPGA clock at 125MHz and the START signal from the master board. Before starting taking data, one needs to configure each ACM board to accept or not an external FPGA clock and the wanted mode: standalone, master, slave. This is done with a set of commands that are grouped into a library in the software operation, see section 2.4 (paragraph shared libraries).

2.2 Software

2.2.1 Configuration

Board ID The board are identified once for all. They are labeled 1XX from 101 to 152 and this number sets internally their identity. Their IP address is set to 192.168.1XX.5. (This is done in the firmware with a look up table with the chip ID of the FPGA one one column and the ID on the other. This table is loaded at power up)

Voltage configuration The voltage configuration is set in a text file with bcf (board configuration) extension. One can set set here the values of bias and clock rails to be loaded. Such bcf file is loaded at the start up procedure. The values that can be set are:

- `VDD1, VDD2`
- `VR1, VR2`
- `VDRAIN1, VDRAIN2`
- `VSUB`
- `Vi_j_L, Vi_j_H, Hi_j_L, Hi_j_H` (i in [1,2,3], j in [A,B])
- `RGi_L, RGi_H, DGi_L, DGi_H, SWi_L, SWi_H` (i in [1,2])

One can also group some settings together:

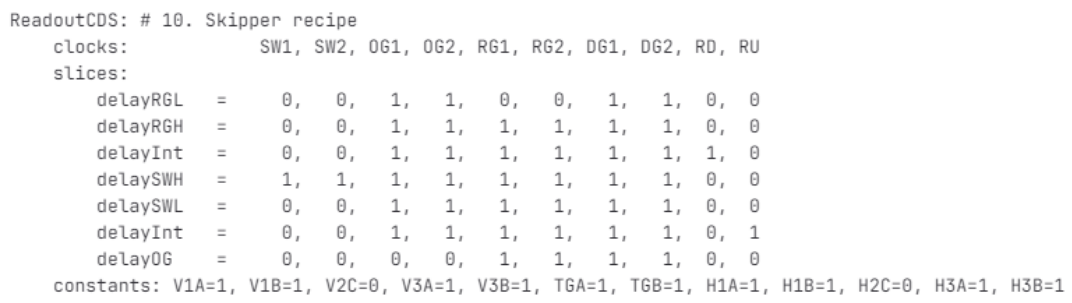
- `VR : VR1, VR2`
- `VDD : VDD1, VDD2`
- `VDRAIN : VDRAIN1, VDRAIN2`
- `H_L` : all the low rail of horizontal
- `H_U` : all the upper rail of horizontal
- `V_L` : all the low rail of vertical
- `V_U` : all the upper rail of vertical
- `RG_L, RG_U` : low and upper rails of the 2 RG
- `SW_L, SW_U` : low and upper rails of the 2 SW
- `DG_L, DG_U` : low and upper rails of the 2 DG

See appendix A for an example
One can change manually those values one by one without the need of the bcf file.

Useful functions:

- `daq.startup_ccd()` **: executes the startup procedure with the voltages given in the bcf file.**
- `daq.set_voltage(key, val)` **: changes voltage**

```
daq.set_voltage('VR2', -6)
```



```
ReadoutCDS: # 10. Skipper recipe
    clocks:            SW1, SW2, OG1, OG2, RG1, RG2, DG1, DG2, RD, RU
    slices:
        delayRGL   =     0,   0,   1,   1,   0,   0,   1,   1,  0,  0
        delayRGH   =     0,   0,   1,   1,   1,   1,   1,   1,  0,  0
        delayInt   =     0,   0,   1,   1,   1,   1,   1,   1,  1,  0
        delaySWH   =     1,   1,   1,   1,   1,   1,   1,   1,  0,  0
        delaySWL   =     0,   0,   1,   1,   1,   1,   1,   1,  0,  0
        delayInt   =     0,   0,   1,   1,   1,   1,   1,   1,  0,  1
        delayOG    =     0,   0,   0,   0,   1,   1,   1,   1,  0,  0
    constants: V1A=1, V1B=1, V2C=0, V3A=1, V3B=1, TGA=1, TGB=1, H1A=1, H1B=1, H2C=0, H3A=1, H3B=1
```

Figure 4: Example of the definition of a sequencer function. In this case, the function used to perform a NCDM. The moving clocks during this function are SW1, SW2, OG1, OG2, RG1, RG2, DG1, DG2, RD, RU, the other are defined as constant. The names delayRGL, delayInt etc are time constants defined at the beginning of the file.

Sequencer The sequencer file contains the definition of the time sequences of the CCD clocks. The sequencer is a entire firmware block, but it has also a representation in the software that the user has access to. The user describes the various sequences in a text file (.seq) with a defined syntax. It is then compiled in the LDAQ and sent to the FPGA.
The sequencer file is split in several parts: the constant definition, the signal definition (one shouldn't touch as it is related to the firware and hardware), the function description, and the subroutine definition. One can create function as a two entries tables associating the binary state (0 or 1) of a signal for a given time period (one time slice). For example the function in Fig 4 shows the readout of one pixel (1 NDCM) where the signals RU (Ramp Up) and RD (Ramp Down) are indicative of the time where the video signal is digitally integrated. Functions are used as building block for more complex sequences which are described as subroutines. The number of timeslice in each function has to be lower than 16. Each time slice is defined with a 10ns granularity. In the DAQ, one will point to a given subroutine and then sends a START to the board which will make the firmware to run the time sequence. Once the time sequence is started, there is no way to stop it.

The documentation of the firmware implementation and the human readable description language are given in [3][1].

Useful functions:

- `daq.init_ccd("Module_LPNHE.bcf","LPNHE_module.seq")` **: sets the bcf and seq files**
- `daq.set_seq_var(key, val)` **: changes sequencer variable**
- `daq.load_current_sequencer()` **: compiles and uploads the sequencer**

Configuration logging :
- configuration files:
The configuration set by the user is in the base information files XX.bcf and XX.seq files. However this base information can be changed on the fly. Hence files `XX_current.bcf` and `XX_current.seq` are created when the base files are loaded. The current files are the ones that are updated when the configuration is changed. When an image is acquired the `XX_current.bcf` and `XX_current.seq` files are saved under the folder `config/configfiles (or seqfiles)/saved/` with the hash as a second extension.
- Metafile
Upon the image acquisition, a metadata file is created it contains:

- the time of the acquisition start and end
- the configuration related to the bcf file (biais and clock voltages)
- the timing constant of the sequencer
- the name and the hash of the sequencer

- .pkl file
The LDAQ object is saved into a pickle file when the python shell is exited. As the full object is dumped into

[1]In the documentation two functionalities are not working: the infinite loop with subroutine, and the pointer to subroutine.

this file, one can restart the DAQ taking this file as base. (This is usefull in case of DAQ crash, as if you restart from scratch, there is a mismatch of status between the hardware/firmware and the software)

Data format and decoder The 4 ADC (LTC2387) are 15MS/s 18bits. They are read out continuously by the firmware and the user can choose the level of data to output.

- level 0: raw data, can be output only on 1 channel
- level 1: skip data: the sum and the sum of the square of each signal and pedestal samples (defined by the RU and RD signal in the sequencer) are included
- level 2: pixel data: the sum and the sum of the square of the Correlated double sampled (pedestal - signal) data are included.
- level 3: mute: no data at all

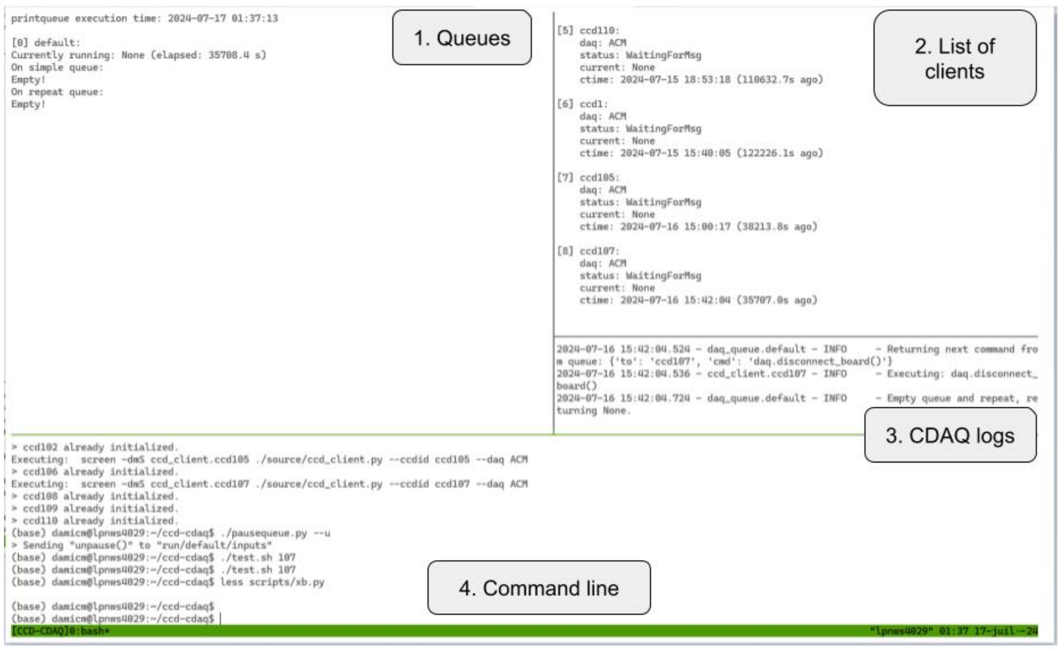
Useful functions:

- `daq.ops.acquire.set_readout_mode("acq2")` **: sets the level of acquisition**

-decoder:
As the data primarily written are in a binary format a decoder allows the translation into usual format such as FITS for the images and .csv for the raw data. The decoder is run automatically if the process `image_builder.ACMDecoder` is started at the acquisition time. One can also do it manually by loading the library under `/acmpy/decoderlib/decoder.py`.

2.3 CDAQ operation

Here we give a bit of details on how to operate the CDAQ. Again, it is well explained in the Readme on github so please go visit `https://gitlab.com/nicolaseavalos/ccd-cdaq`.
The CDAQ is a TMUX window with 4 panes (see Fig 5).



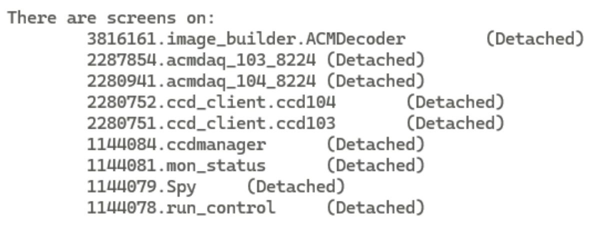
```
printqueue execution time: 2024-07-17 01:37:13

[0] default:
Currently running: None (elapsed: 35708.4 s)
On simple queue:
Empty!
On repeat queue:
Empty!
```

```
[5] ccd110:
    daq: ACM
    status: WaitingForMsg
    current: None
    ctime: 2024-07-15 18:53:18 (110632.7s ago)

[6] ccd1:
    daq: ACM
    status: WaitingForMsg
    current: None
    ctime: 2024-07-15 15:40:05 (122226.1s ago)

[7] ccd105:
    daq: ACM
    status: WaitingForMsg
    current: None
    ctime: 2024-07-16 15:00:17 (38213.8s ago)

[8] ccd107:
    daq: ACM
    status: WaitingForMsg
    current: None
    ctime: 2024-07-16 15:42:04 (35707.0s ago)
```

```
2024-07-16 15:42:04.524 - daq_queue.default - INFO     - Returning next command fro
m queue: {'to': 'ccd107', 'cmd': 'daq.disconnect_board()'}
2024-07-16 15:42:04.536 - ccd_client.ccd107 - INFO     - Executing: daq.disconnect_
board()
2024-07-16 15:42:04.724 - daq_queue.default - INFO     - Empty queue and repeat, re
turning None.
```

```
> ccd102 already initialized.
Executing:  screen -dmS ccd_client.ccd105 ./source/ccd_client.py --ccdid ccd105 --daq ACM
> ccd106 already initialized.
Executing:  screen -dmS ccd_client.ccd107 ./source/ccd_client.py --ccdid ccd107 --daq ACM
> ccd108 already initialized.
> ccd109 already initialized.
> ccd110 already initialized.
(base) damicm@lpnws4029:~/ccd-cdaq$ ./pausequeue.py --u
> Sending "unpause()" to "run/default/inputs"
(base) damicm@lpnws4029:~/ccd-cdaq$ ./test.sh 107
(base) damicm@lpnws4029:~/ccd-cdaq$ ./test.sh 107
(base) damicm@lpnws4029:~/ccd-cdaq$ less scripts/xb.py

(base) damicm@lpnws4029:~/ccd-cdaq$
(base) damicm@lpnws4029:~/ccd-cdaq$ |
[CCD-CDAQ]0:bash*                                          "lpnws4029" 01:37 17-juil.-24
```

Figure 5: Example of one CDAQ window.



```
There are screens on:
        3816161.image_builder.ACMDecoder        (Detached)
        2287854.acmdaq_103_8224 (Detached)
        2280941.acmdaq_104_8224 (Detached)
        2280752.ccd_client.ccd104       (Detached)
        2280751.ccd_client.ccd103       (Detached)
        1144084.ccdmanager      (Detached)
        1144081.mon_status      (Detached)
        1144079.Spy     (Detached)
        1144078.run_control     (Detached)
```

Figure 6: example of the screen list one should find when operating two boards

start up Once in the directory `/ccd-cdaq/` one should run

```
./start_daq.sh
python start_imgbuilders.py
python ./start_client.py
```

This will start the TMUX window and run the image builder in background. One can then start writing commands in `Command line` pane, they will fill the queues. The list of the commands will appear in the `Queue` pane and disappear as they are executed. The clients defined in `/experiment/ccd_clients.json` are started which means an LDAQ object was created for each of them in a screen session. The list of clients and the output of the CDAQ logs are shown in the panes 2 and 3.

commands The syntax to send command to the client is the following:
`./sendrun.py --to CCDID "CMD"` : where CCDID = ccd1XX (XX from 01 to 52) and CMD is for instance `daq.startup_ccd()`. One can also run python scripts of such commands. Please see section2.4 for the actual CCD operation.

CCD pool In order to send the same command in parallel to several CCD systems you can organise the CCD system in pools. You can inlcude one CCD system to the pool with the command:
`./sendrun.py --to ccdid "listen2pool('ccdpool')"`
and then to send command to the pool the ccdid becomes ccdpool for instance:
`./sendrun.py --to ccdpool "daq.startup_ccd()"`

shut down There is no reason to shutdown the complete CDAQ after it was started but client can be shutdown.To shut down clients of the CDAQ:

```
./shutdown_client.sh ccdid
```

This will kill the client session and delete the LDAQ object.

list of screen An example of the list of screen (`screen -ls`) one should see when operating is shown in Fig. 6

2.4 CCD operation

2.4.1 main operation

initialisation: Affects the configuration bcf and sequencer files to the LDAQ object.

- `daq.init_ccd(XX.bcf, XX.seq)`

startup CCD: Configures the bias and clock voltages to the initial bcf file value.

- `daq.startup_ccd()`

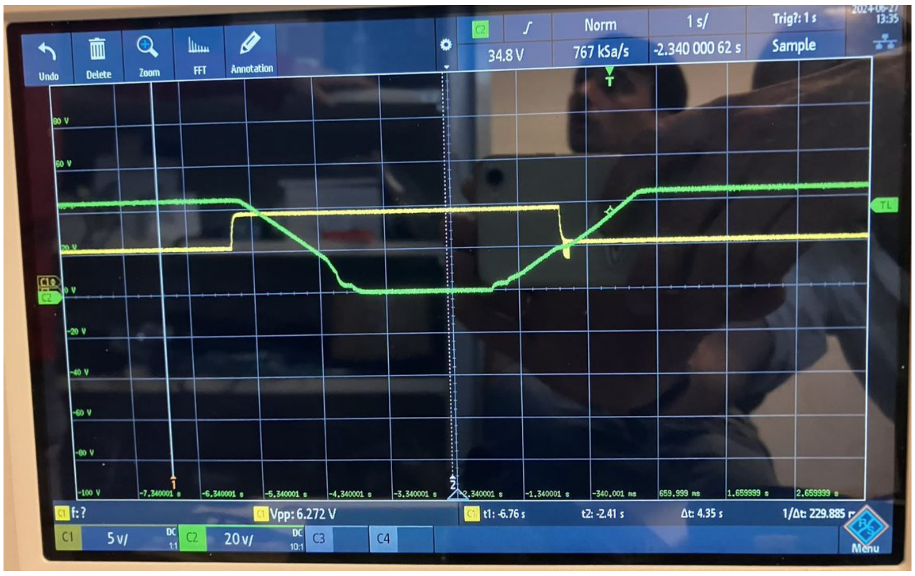


Figure 7: VSUB and one vertical clock as the erase procedure is performed

erase/epurge CCD Procedes to an erase procedure. The VSUB and vertical clock during the erase procedure are shown in the Fig. 7

- `daq.ops.erase_CCD(highP=9., delay = 2)`
- `daq.ops.epurge_CCD(lowP=-9., delay = 1)`

image acquisition Prepare the image by loading the sequencer is needed, open the metadata file, and sends a start signal to the board. The data can be acquired in level 2 or level 1.

- `daq.prepare_image(fname="img", level=2)`
- `daq.only_take_image(level=2)`

PSD acquisition Set the acquisition level to level 3, starts a static sequencer (no clock are moving) and record raw data on one channel.

- `daq.take_psd(fname="PSD", ch=channel)` **channel in 0,1,2,3**

Trace acquisition Set the acquisition level to level 3, starts the sequencer and record raw data on one channel. The total time length of the sequencer must be short ($<$ 1s) otherwise the data ouptut will be too large.

`daq.take_trace(fname="trace", ch=channel)` **channel in 0,1,2,3**

infintite flush The flush is done by repeating infinitely a sequencer that flushes a small number of lines. When you want to stop the infinite flush it might take a few tens of seconds in order to finish the current flush.

```
daq.start_infinite_flush()
```

Those operation are combined in scripts provided with this documentation. From the previous explanation, you should be able to understand them and write your own if needed. Here is a quick description of each script:

2.4.2 shared libraries

_damic.py

GetCCD : allows one to call the script described below with their ccdid. For instance `python start_module.py ccd105`
Send, SendR: wrapper arround the send command (no need for the id), for instance `Send('daq.startup_ccd()')`.
The SendR is for the Repeat queue.

_synchrolib.py It contains two functions to set the mode of synchronisation.

- standalone_readout: set the internal clock, and the standalone mode
- synchro_readout: set the external clock, and one client to master and the other to slave mode

This library was written for 2 CCDs. It can be easily extended to more. The ccdid are hardcoded, so one should modify this code according to the hardware.

2.4.3 main scripts

start_module.py connection to client, voltage startup procedure, and erase epurge. Should be done only once at the beginning of the operation.

take_psd_standalone: acquisition of raw data with a static sequencer (noise evaluation) (BEWARE: you should not do that with more that board at a time)

take_image_standalone.py: acquisitioon of an image with boards running independently (standalone mode)

take_image_synchro.py: acquisition of an image in fully synchronised mode.

end_run.py: decrease the voltages, and shuts down properly the LDAQ.

3 Miscellanuous remarks

PSD, traces Again: do not do that at the same time with several boards (one board will spit out 600MBits/s so any 1GBit/s network will be fine with one board but not with 2 !)

in case the DAQ freezes In case the DAQ freezes one can restart smoothly by doing the following:

- `python shutdown_client ccdid`
- check that the screen sessions (`ccd_client.ccdid` and `acmdaq_ccdid_8224`) are both killed.
- `./start_client.py`
- `./sendrun.py --to ccdid "daq.restart_board(ccdid)"`

References

[1] Samtec to dsub adapter schemtics and fabrication documents. URL https://gev.uchicago.edu/cgi-bin/DocDB/ShowDocument?docid=1148.

[2] Acm schematics: Vme connector. URL https://edg.uchicago.edu/~bogdan/DAMIC_ACM/doc/sch_pdf_files/Sch3018_9.pdf.

[3] Lsst sequencer documentation. URL https://gev.uchicago.edu/cgi-bin/DocDB/ShowDocument?docid=1149.

[4] Quartus software for linux. URL https://www.intel.com/content/www/us/en/software-kit/666220/intel-quartus-ii-web-edition-design-software-version-13-1-for-linux.html.

A Example of bcf file

```
    ; Configuration file

[board]
host = localhost
; port to communicate with the odaq server
port = 8224
; identification of the electronics system (related to Odile board)
board = 0
ipaddress = 192.168.1.5
; ip address of the odile board
server_ipaddress = 192.168.1.1
; ip address of the PC ethernet connection
datalink = copper
; copper or optical


;===============================================================================
[channel_map]
; ext1 = ch0 is 1A with DB50, 1B with only samtec
; ext2 = ch1 is 2A with DB50, 2B with only samtec
; ext3 = ch2 is 1B with DB50, 1A with only samtec
; ext4 = ch3 is 2B with DB50, 2A with only samtec

; on normal module 1A->A, 2A->B, 2B->C, 1B->D
; if flipped       1A->B, 2A->A, 2B->D, 1B->C
ch0 = CCD_A
ch1 = CCD_B
ch2 = CCD_D
ch3 = CCD_C




;===============================================================================

[readout]
; mostly acquisition related parameters
; FW filter mode: 0: raw, 1: pixel level, 2: skip level, 3: mute
FILTER0 = 3
FILTER1 = 3
FILTER2 = 3
FILTER3 = 3

;===============================================================================
[bias]
; values of biases in V
VDD1    = -18.
VDD2    = -18.
VR1     =  -7.
VR2     =  -7.
VDRAIN1 = -23.0
VDRAIN2 = -23.0
VSUB    = 60.

;===============================================================================
[clock_offsets]
OFS_RS1 = -10
OFS_RS2 = -10
OFS_CLK = -10
```

```
[clock_rails]
V1_A_H = 9.
V1_A_L = 7.
V1_B_H = 9.
V1_B_L = 7.
V2_C_H = 9.
V2_C_L = 7.
V3_A_H = 9.
V3_A_L = 7.
V3_B_H = 9.
V3_B_L = 7.

TG_A_H = 7.
TG_A_L = 7.
TG_B_H = 7.
TG_B_L = 7.

H1_A_H = 8.
H1_A_L = 5.5
H1_B_H = 8.
H1_B_L = 5.5
H2_C_H = 8.
H2_C_L = 5.5
H3_A_H = 8.
H3_A_L = 5.5
H3_B_H = 8.
H3_B_L = 5.5

SW_1_H = -3
SW_1_L = -10
OG_1_H = -4.
OG_1_L = -9.
RG_1_H = 7.
RG_1_L = 2.
DG_1_H = -4.
DG_1_L = -10.

SW_2_H = -3.
SW_2_L = -10.
OG_2_H = -4.
OG_2_L = -9.
RG_2_H = 7.
RG_2_L = 2.
DG_2_H = -4.
DG_2_L = -10.
```

B TMUX commands

- `ctrl-b d` detach window
- `tmux a` attach window (if only one)
- `tmux attach -t session-name` attach window
- `tmux ls` list windows